AD-A207 006

(4)

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NW-LIS-89-23-02	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Semiannual Technical Report No. 2 "VLSI Architectures & CAD"		5. TYPE OF REPORT & PERIOD COVERED Technical, Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Northwest Laboratory for Integrated Systems		8. CONTRACT OR GRANT NUMBER(s) N00014-88-K-0453
9. PERFORMING ORGANIZATION NAME AND ADDRESS Northwest Laboratory for Integrated Systems University of Washington Dept. of Computer Science, FR-35 Seattle,WA 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA-ISTO 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE April, 1989
		13. NUMBER OF PAGES 43 pages
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research - ONR Information Systems Program - Code 1513: CAF 800 North Quincy Street Arlington, VA 22217		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE 11 APR 1989 S D E

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

NW LIS, VLSI, CMOS, CAD, APEX, Wirelisp, OHMICS, Parallelism, Logic Arrays, Simframe, Multimode Simulation

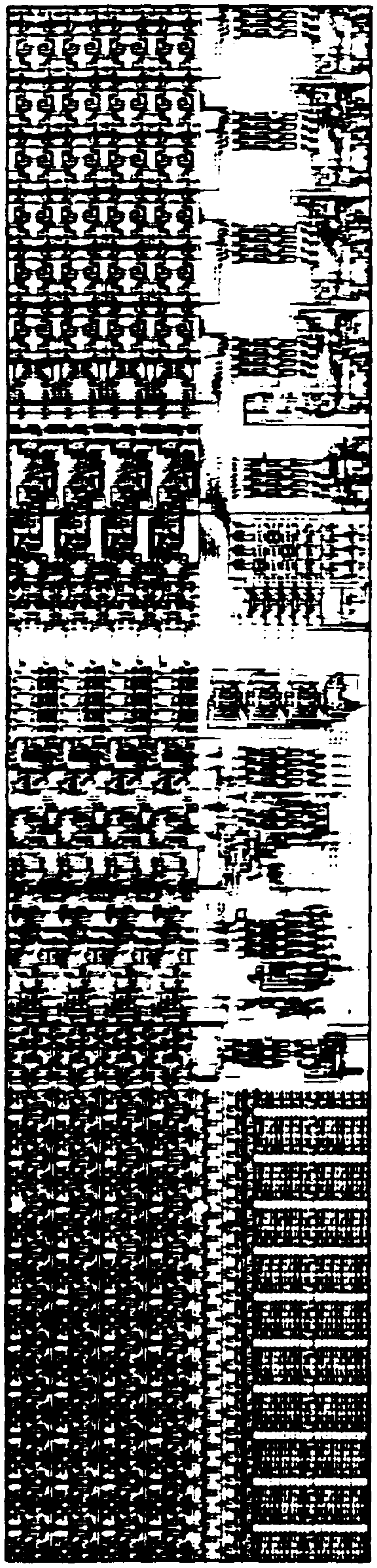
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document reports on the research activities of the Northwest Laboratory for Integrated Systems, for the period of November 15, 1988 to April 3, 1989, under the sponsorship of the Defense Advanced Research Projects Agency, and the Office of Naval Research, under contract number N00014-88-K-0453.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



NORTHWEST LIS
(LABORATORY FOR INTEGRATED SYSTEMS)

Semiannual Technical Report No. 2
"VLSI Architectures & CAD"
University of Washington

April 3, 1989
TR #89-23-02

Reporting Period: November 15, 1988 - April 3, 1989

Principal Investigator: Lawrence Snyder

Sponsored by:
Defense Advanced Research Projects Agency (DoD)
Issued by Office of Naval Research
Under Contract #N00014-88-K-0453

Contents

1 Overview of Activities 2

2 Bit Serial vs Word Parallel Computers 3

3 Development of Wirelisp 4

4 Simframe: A Framework for Multimode Simulation 5

5 Behavioral Representation of Digital Systems 6

6 Timing Optimization of Multi-phase Sequential Logic 6

7 Adaptive Scripts for MIS 7

8 A Demonstration of the APEX Architecture 8

9 A Hardware Data Compressor 9

10 Reconfigurable Logic Arrays 10

11 Work on a Low Cost Testing Environment 11

Appendices

A Are Bit Serial or Word Parallel Computers Faster? . . . .

B Wirelisp: Graphical Programs for Digital System Design . . . .

C Ohmics: A Program to Check Ohmnic Contacts . . . .

→Keywords: Integrated
Su[illegible], Very Large Scale Integration,
Computer aided design, Computer architecture,
Complementary Metal Oxide Semiconductors,
[illegible] optimization
[illegible], Multilevel [illegible] [illegible]ation

1

1 Overview of Activities

Current research in the LIS is focused in three areas - architecture studies, general techniques for IC CAD, and application-specific circuit designs.

A recent study of bit serial and word parallel computer architectures has yielded some interesting information on the utility of each in various problem realms. The primary result is a strong argument in favor of a mean between the extremes of large numbers of bit serial processors and small numbers of processors each with a wide datapath.

For some time we have investigated an architecture for efficient curve drawing. This work has led to two different chip implementations - APEXI and APEXII. Recently we received functional chips of APEXI and are now designing a board-level interface to the Nu Bus of an Apple Mac II. When completed, this system will be a prototype solids modeling workstation based on the APEX architecture.

Recently we began investigating various data compression algorithms with characteristics amenable to hardware implementation. This work has so far yielded a variant of the Lempel-Ziv algorithm that adapts continuously to its input and is appropriate to a hardware implementation with fixed memory.

Several efforts have been pursued in the area of design specification and optimization. Wirelisp is a language for specifying circuit structure that allows graphical and procedural constructs to be mixed at a fine-grained level. Also in this area is an effort to formulate a behavioral representation of digital circuits. This representation is novel in that it unifies the behavior and structure and includes timing behavior and constraints.

Several design optimization projects deal with extensions to MIS, the Multilevel Logic Minimization tool developed at UCB. One project is to develop adaptive scripts to control the minimization search. Another attempts to introduce timing constraints into the minimization process, and find an optimal distribution of logic between phases of a multiphase sequential circuit.

In the area of design verification, we have two efforts. One is the development of a framework for multimode simulation. Another is a tool (OHMICS) for analyzing substrate connectivity in CMOS designs and reporting regions of substrate that are insufficiently tied to the appropriate power rail. (Described in Appendix C.)

We continue our distribution of VLSI design tools. Since September 1988, when Release 3.2 first became available, we have distributed over 43 copies.

2 Bit Serial vs Word Parallel Computers

(Sam Ho, Larry Snyder)

The word size of a computer, or the size of the largest number that the computer can handle in one operation, is one of the most basic parameters of a computer. Over the years, microprocessors have grown from four bits to eight, sixteen and thirty-two bits. Meanwhile, larger computers have used thirty-two, thirty-six, and sixty bits, among others, as a word size. The number of processors is another crucial measure of a computer. It is common for computer manufacturers to offer additional processors as attachments to improve the performance of an existing computer. Some designers take the view that it is best to maximize the number of processors in a computer at the expense of using a narrower processor. This view is typified by the Connection Machine, which has 65,536 processors, each capable of handling only a single bit.

The goal of this research is to determine the relative merits of a small number of high bandwidth processors versus a large number of low bandwidth. correct, if any. To do this, we reduced computation to the act of transferring information between the processors and memory on the principle that any other computation would be affected proportionately. We also assumed that because of latency in the memory system, there is some delay associated with this activity during which some other references might be processed. Then we examined the ways to convert from one processor to another, subject to the condition that the product of the number of processors and the width of each is held constant. In this way, the effect of the width is separated from any change in the overall size of the computer system.

The result is that neither extreme is optimal, but an intermediate range is best. Overly wide processors are unable to take advantage of the excess width, while overly narrow (and numerous) processors are unable to take advantage of excess processors. Furthermore, the intermediate range of best performance is more than a single point. At any point within a substantial region, changes in processor width are exactly offset by the corresponding change in the number of processors. Analysis and results are described further in Appendix A.

3 Development of Wirelisp

(Zhanbing Wu, Carl Ebeling)

Wirelisp is a language used to describe the structure of a circuit. The structure can be described both graphically and procedurally where lisp expressions may be included in circuit drawings and circuit drawings may be included in lisp expressions. A number of features such as structured signals, iterators and optional parameters make the language very expressive. A complete description of the language is included in Appendix B.

The first version of a Wirelisp environment has been operational for about six months. This environment includes the graphical editor XDP for drawing Wirelisp programs, a backend analyzer which converts these drawings into Wirelisp, and a Wirelisp interpreter written in T. This system has been used to design the Reprogrammable Logic Array chip described in section 10.

We have recently finished work on several enhancements. The most important is the ability to reference arbitrary behavioral descriptions. Behavioral descriptions are represented as modules for which the input-output behavior is specified explicitly, for example using logic equations or finite state machines. These modules reference both a behavioral description and a method by which the structure implementing that behavior can be generated. This implementation can vary depending on the particular output desired. For example, the implementation may be a functional model for COSMOS or Network C, a multi-level circuit generated by the MIS tools or a PLA. The overall goal of this work is to provide a simple framework in which the designer can specify and evaluate a system at a level appropriate to the level of detail required.

We have modified the graphical editor XDP so that it works under X version 11. A number of enhancements are planned to allow the efficient editing of hierarchical circuit descriptions and text.

4 Simframe: A Framework for Multimode Simulation

(Craig Anderson, Mary Bailey, Gaetano Borriello, Larry McMurchie)

In the process of designing a circuit, one uses a variety of simulation models varying in the level of abstraction they employ. Typical model types include iterated timing analysis, switch level RC delay, switch-level unit delay, and boolean functional models. The choice of a model depends on the characteristics of the individual subcircuit that is being modeled. Simulating the entire circuit at a particular level may not be possible if the functionality of a device is not adequately described at that level. This may occur when a mostly digital circuit has some analog character, preventing the use of a switch-level model to be used on the entire circuit. At the same time, the size of the circuit may prevent the use of analog models for all the devices.

This classic problem illustrates the need for a multimode simulator, one that allows the use of varying levels of models simultaneously. Our approach to the problem is "Simframe", an event-driven framework that supports a variety of models. The framework revolves around the general notions of events and data types. Data types are determined by the the type of model being used. An important point is that data types are not hard-wired into the event handling mechanism; they are dependent on the characteristics of user-defined models.

A major problem to be solved in any multimode simulation scheme is how to handle the conversion between data types at the interfaces between models. A typical conversion one needs to make is between the switch-level resistance/voltage pairs and analog current/conductance pairs. Although this Thevenin/ Norton transformation is straightforward, it may be complicated by the presence of unknown resistances and voltages in the switch level state description. An additional complication is bidirectionality of an interface and the need for an iterative scheme to settle on the state of the circuit.

Another problem to be dealt with is the concept of time. How does one translate a unit-delay switch-level simulator's notion of time into the real time of an analog simulator or the user-specified delays of a functional simulator?

Thus far, we have looked at several simulation models and proposed specifications for the data conversion between them as well as schemes for the synchronization of individual clocks. Included in our survey are the switch level unit delay models of MOSSIM II and COSMOS; the switch level RC delay model of RNL; the iterated timing analysis algorithm used in SPLICE3; and the functional circuit description of

Network C. Current work focuses on the specific problem of extracting the necessary voltage and path resistance information from the MOSSIM II algorithm.

5 Behavioral Representation of Digital Systems

(Tod Amon, Gaetano Borriello, Rick Hood, Wayne Winder)

Our research involves developing a new behavioral representation for digital systems. After we develop a working representation we intend to develop general algorithms for the synthesis of structure (hardware) from our representation. Our representation for behavior is novel in that it unifies the representations of behavior and structure and includes timing behavior and constraints. It is a low level representation; we envision the existence of a variety of CAD tools which compile high level behavioral descriptions (timing diagrams, HDL, FSMs, etc) into our representation. Additional CAD tools would then synthesize hardware using our representation and optimize for a variety of different needs.

We are currently in the process of completing the syntax and semantics of our proposed representation. We have a working formal description of its syntax and semantics, and are developing an EDIF (Electronic Design Interchange Format) ASCII file representation which will be used as input to synthesis algorithms and can be derived from high level languages which compile into our representations.

We have developed a representation for the behavior of digital systems and need to validate it by developing algorithms and tools which work with the representation. Our first step will be to demonstrate that a series of transformations applied to the representation will result in the synthesis of valid structure. This work will require the development of new algorithms for behavioral synthesis, as our representation allows for a very general class of behavioral specification and includes complex timing constraints.

6 Timing Optimization of Multi-phase Sequential Logic

(Karen Bartlett, Gaetano Borriello, Sitaram Raju)

Timing optimization is used to reduce the cycle time of a synchronous system. In the case of complex integrated circuits, the design often uses multi-phase clocking methodologies. Most current logic synthesis and optimization systems only optimize the logic between register or latch boundaries for single-phase systems. Retiming algorithms exist to relocate registers and form an equivalent circuit with the shortest cycle time [1]. However these algorithms are not optimal for multi-phase logic and are only applicable when all phases are of the same width. Recently, optimization approaches have been proposed that move logic across register boundaries [2, 3]. However, these approaches use retiming algorithms as a subroutine and therefore cannot support general multi-phase logic.

Our scheme is oriented towards systems with two or more clock phases. The phases are not required to be of the same duration. Input and output signals are specified as being valid or required to be valid during any subset of the phases available.

We are using an existing logic synthesis system (MIS) to perform timing optimization on the logic and define the time for each phase. Our algorithms first move logic off the critical paths and across latch boundaries so as to reduce the total cycle time. This is then followed by an area optimization step in which logic is merged to reduce area as long as the critical path logic can still be implemented within the same cycle time.

[1] . C.E. Leiserson, F.M. Rose, and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Proceedings of the Third Caltech Conference on VLSI*, March 1983.

[2] . S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", *Proceedings of International Workshop on Logic Synthesis*, May 1989.

[3] . G. De Micheli, "Synchronous Logic Synthesis", *Proceedings of International Workshop on Logic Synthesis*, May 1989.

7 Adaptive Scripts for MIS

(Gaetano Borriello, Robert Condon)

MIS (the Berkeley multi-level optimization system) provides a set of operations useful in minimizing a multi-level network of combinatorial logic. The normal method of using these operations is by applying them in a predetermined order (through "scripts"). We have been investigating adding backtracking and conditional execution of the commands in a script.

To this end, we have written a T language interface to MIS. The T interface allows commands to iterate, conditionally execute or to backtrack.

Preliminary investigation indicates that the MIS standard script does reasonably well if the implementation technology is simple standard cells or gate matrix layout. We are currently investigating the use of MIS for more constrained technology mapping. Examples include the new PLDs of non-MOS technologies such as ECL.

8 A Demonstration of the APEX Architecture

(Gaetano Borriello, Tony DeRose, Brian Lockyear, Al-Ping Bien)

APEX is an architecture that facilitates efficient drawing of curves and surfaces from sets of control points (see Appendix B of LIS Semiannual TR, November, 1987). We recently completed testing of MOSIS fabrications of APEXI, one implementation of this architecture. 2 out of 30 chips were functional.

We are currently developing a board design to interface the chip with a Macintosh II computer. The board will host three APEX chips, one for each coordinate axis. Additionally, peripheral circuitry for host interfacing and coordinate transformations will be resident on board. The finished prototype will provide real-time point coordinate streams for display on a stereo monitor.

To date this project, on which we began efforts Winter term 1989, has focused on design and simulation of circuitry required to translate the 16 bit x, y, and z coordinates produced by the trio of APEX chips into a pair of x, y coordinates suitable for display upon a stereo monitor system. In order to avoid a system bottleneck, we currently hope to perform this translation in hardware using some form of programmable logic device.

Design of the translation logic has allowed us to experiment with the development of significant logic structures using the Berkeley Bdsyn logic language. The equations required for the translation have been coded into the Bdsyn language, the programs "compiled" and converted into a format suitable for simulation using UW's RNL program.

The demonstration board will be built around an Apple Macintosh Coprocessor Platform. This board hosts a 68000 capable of running as master of the Macintosh internal Nu-Bus. The 68000 will be responsible for coordinating activities on the APEX board and for high speed transfer of data to the stereo frame buffers. The speed of the board will be limited by the bus speed and not the APEX chips.

9 A Hardware Data Compressor

(Gaetano Borriello, Suzanne Bunton, Richard Ladner, Ken Whaley)

The goal of this project is to construct a hardware data compressor with the following properties: it must consume one input every "cycle", where a cycle is bounded by 100ns; an "input" must be as wide as possible, our ultimate goal being 128 bits or more; it must effectively compress all types of data (i.e. the algorithm must be universal); the implementation must be containable on a small board.

There are essentially four existing types of data compression methods amenable to our needs: Huffman, Arithmetic, BSTW, and Lempel-Ziv. The two algorithms with the most modest memory and computational requirements, also happen to be based upon a dynamic data structure for maintenance of a dictionary of words which have been seen recently. These are BSTW and Lempel-Ziv. BSTW is recent and hitherto unimplemented, as revealed by its authors and the literature; during the next six months we will be looking at the feasibility of a hardware implementation of this algorithm. Lempel-Ziv has been implemented both in hardware (HP's chip, MAGIC) and software (UNIX "compress") and is well-known for its raw speed and effectiveness on all types of data.

The UNIX "compress" implementation is based upon Terry Welch's 1984 variation of the Lempel-Ziv method (LZW). One flaw lies in the fact that the dictionary ceases to adapt after it fills up. It is important that a universal hardware data compressor adapt to the input stream continually, since the memory size is fixed and input streams can be arbitrarily long. A long input stream in which the first several words are not representative of the rest of the data will cause LZW to perform poorly. Our theoretical result is a new dynamic data structure which enables the dictionary of a Lempel-Ziv scheme to adapt continually, thus avoiding this problem entirely. At this point the algorithm design using this adaptation scheme is correct and complete. Compressor and decompressor simulators have been completed and reflect register transfer descriptions of the circuits. Proofs of worst case performance have been outlined. The simulators have been used on a small scale to compare the performance of our algorithm to LZW (it's at least as effective, using the same size of dictionary). The hardware implementation is underway and is described below.

Our Lempel-Ziv data compressor consumes one k-bit input every 100 ns, where k can be as large as 8. For $k < 6$, the design fits on a 8x9mm chip. Best average compression results are obtained when the output channel is between 1.5 and 2 times as wide as the input channel. The memory requirements are roughly 2^w words, where w is the output channel width. The design is very simple and consists of 2 dual-ported CAMs and 8 registers with only a handful of nand gates implementing the control. The key features of the design are the CAMs. CAM1 must support simultaneous matches and writes, and CAM2 must support simultaneous reads/writes as well as match writes. All hazards due to simultaneous operations to one cell are avoided via control logic external to the memory. Layouts of both CAM cells have been done according to MOSIS SCMOS rules and simulated using SPICE. The size and speed of the cells are well within our requirements. The rest of the circuit remains to be constructed, though there is plenty of area for registers, counter, logic, busses, precharger logic, and decoders.

10 Reconfigurable Logic Arrays

(Bill Barnard, Robert Condon, Carl Ebeling)

Recently work was started on a Re-programmable Logic Array chip. While the chip is intended for generic pattern matching applications, the initial design application is geared towards position evaluation in chess. This project has also served as a test vehicle for some of the tools we have developed over the past two years, most notably Wirelisp and WIN.

The current status of the project is as follows:

- The entire RLA chip has been specified using Wirelisp and simulated using RNL and COSMOS.
- Layout generators have been written using WIN for all the modules in the chip. WIN is a language developed at the University of Washington which forms the basis of a structured environment for design generator development.
- The layout generated by the WIN module generators has been validated using Gemini. The placement of modules was done manually using Magic and the majority of the routing done using the Magic router.
- The key building block of the RLA chip, a programmable PLA with 8 inputs, 8 outputs and 32 product terms, has been successfully fabricated and tested.
- The complete RLA chip is currently being simulated and the architectural parameters are being finalized. The final routing and verification using Gemini should be completed in April and the chip released for fabrication.

11 Work on a Low Cost Testing Environment

(Carl Ebeling, Neil McKenzie)

We are currently designing an inexpensive low speed tester for use with the MacII. The goal is to produce a high-quality environment for testing and debugging chips and boards for use in hardware and VLSI design classes. Apple is partially funding this effort and hopes to make the tester available to the University community at nominal cost.

The tester will conform to a standard software interface which will be used to integrate simulators, test program languages and display facilities. The goal will be to make the chip under test appear as a module in a simulation so that test programs developed for simulation can be used to test the chip or board as well. The tester will be extensible in terms of I/O signals and we are currently investigating ways to allow higher speed testing as well. We expect to have a prototype tester in operation by Summer and the software environment in place by Fall.

Are Bit Serial or Word Parallel Computers Faster?

Samuel Ho and Lawrence Snyder [*]
Department of Computer Science
University of Washington, Seattle

Keywords: Architecture, Memory bandwidth, Model of parallel computation, Latency.

Abstract

A model of parallel computation based on memory references is presented. The model describes computation as a sequence of references. Within this model two theorems are proved regarding the effect of memory latency, instruction execution time, and memory bandwidth on computation time, and the tradeoffs possible among these parameters are identified. These theorems are then applied to a comparison of bit-serial and word-parallel architectures, answering the question, which is faster?

1 Introduction

When asked why they selected the bit serial or word parallel approach to building parallel computers, architects of each design style have given the same answer, "To best utilize the available memory bandwidth." Both positions cannot be right, of course, unless it doesn't matter. But considering the vehemence with which the proponents advocate each choice, the conventional wisdom seems to be that it does make a difference. This paper inquires into the question, Are bit serial or word parallel computers faster?

Implied in the question is the added condition of, *ceteris paribus* — everything else being equal. Such a condition is necessary because without it, any answer to the question could be achieved simply by controlling factors such as clock speed, amount of hardware, memory latency, technology, instruction repertoire, target programs, etc., and there are already innumerable examples of such apples-and-oranges machine comparisons.

But the *ceteris paribus* condition shouldn't be too quickly embraced because controlling these factors is precisely what the task of computer design is all

*Supported in part by DARPA Contract N00014-88-K-0453. Electronic mail address samho@cs.washington.edu or snyder@cs.washington.edu. Telephone 206-543-9265.

about. There is little benefit in an analysis that obscures the specific advantages of the two types of design by setting all factors equal. For example, if either bit serial or word parallel processors permit a faster clock rate than is achievable by the other, then postulating machines with equal clock rates excludes these faster machines from consideration. Thus, the analysis requires some care to produce an apples-and-apples comparison and at the same time respect the special properties of the bit serial and word parallel processors.

What is known about the benefits of each style? Though there have been some analyses purporting to demonstrate the superiority of bit serial [Hockney and Jesshope 1981] and word parallel [Danielsson 1983] architectures, they are inadequate [Yang 1987]: They are applied only to pipelined uniprocessors and thus ignore considerations peculiar to parallel computers, they are limited to specific examples rather than general computations, and they have produced apparently contradictory results.

In the absence of a definitive analysis, proponents have offered informal reasons to prefer one style over the other.

Several benefits are claimed for the bit serial design. Hockney and Jesshope [1981] argue in effect that there is "less depth" to circuits that process data serially. For example, in a bit serial **add**, the carry is simply accumulated as the sum is being formed, while in a word parallel **add**, carry propagation delays the completion of the computation. Bit serial machines can operate on small data values more efficiently since they avoid packing and unpacking overhead [Hillis 1985, Yang 1987]. They can implement operations more selectively; for example, the "radix" approach to finding the maximum of numbers stored 1 per processor tests the bits in most significant to least significant bit order, masking processors that fail to have a bit set. Finally, transfer of data from memory can be pipelined, covering memory access latency.

Word parallel machines also have benefits. With more data, parallelism can be exploited in the processor when executing the basic instruction operations; this means that during the processing step, which is overlapped with memory operations and whose duration is determined by the time for memory operations, more work can be completed [Danielsson 1983]; for example parallel multipliers can execute **multiply** in one instruction time. This has a secondary consequence of providing an opportunity to use more hardware within the processor element itself; besides a specialized multiplier, floating point circuitry and CISC level instructions are possible. Word-size transmission is a more direct way to implement instruction and address processing, and therefore more appropriate for MIMD computers. Finally, caches are practical for word parallel processors and these can reduce latency in memory reference.

There are, no doubt, other features that can be exploited by bit serial and word parallel computers, but finding them is not the work for the present study. Rather, the problem is to understand how each feature improves the performance of the machine for which it is advocated, and whether there are features available to the competitive architecture that offset this advantage. For this

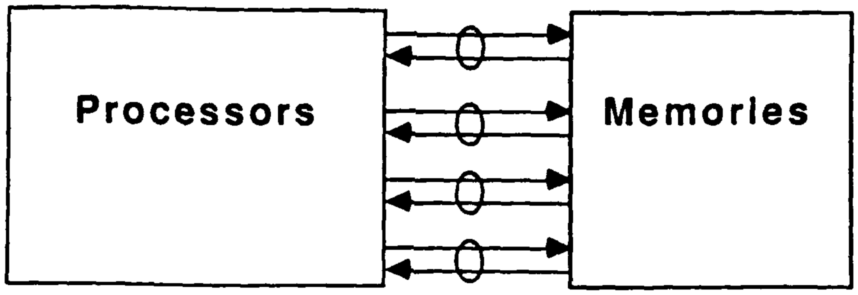


Figure 1: Connection between processors and memories

problem, a model of parallel computation is needed.

2 The Model

The approach to the analysis is as follows. A simple model based on a half dozen parameters will be developed first. This model, though overly simplified, will permit a direct comparison between bit serial and word parallel machines. Then, some of the special features noted above will be incorporated into the analysis. Not all of the features mentioned will be analyzed individually or in groups, but the process is started and the methodology is illustrated.

The computational model focuses on the connection between the processors and the memories, as shown in Figure 1.

2.1 Quantitative Units.

Bit serial and word parallel are the extreme points of a range of data path widths and it is important to include the whole range in the analysis. The *data path width, b,* is the number of bits transmitted simultaneously between a processor and memory in a memory reference. For von Neumann computers the concept is well defined; for parallel computers the concept applies to the processor elements, all of which are assumed to have the same width. Thus, for example, $b = 32$ for the Motorola 68020 [Motorola 1984] and the RP3 [Pfister *et al.* 1985], $b = 16$ for the Cosmic Cube [Seitz 1985], $b = 8$ for the Pringle [Kapauan *et al.* 1984], $b = 1$ for the MPP [Batcher 1980].

Each simultaneous transmission between the processor and memory will be called a *half-step*. Time will measured by the number of steps, where a step consists of two half-steps. We make this doubling so that, in one step, the processor can transmit both an address and a value to or from the memory.

In actual processors, this practice is quite common. Of course, many so-called 32-bit processors actually have 64 wires leading to the memory, with 32 lines dedicated to each of the address and value. These processors, with twice the bandwidth, get twice the information transmitted per unit time and so must be scaled accordingly.

The data path width constrains the rate at which a processor fetches data from memory, and symmetrically, the rate at which values are stored. It also constrains the rate at which addresses can be presented to memory in most cases. The case where it does not is when every processor references the same memory location, i.e. the D_{1m} case in the parlance of the Synchronous Taxonomy [Snyder 1988], and this one address can be transmitted from the controller by a broadcast network. Otherwise, the address is assumed to be transmitted across the same or an equivalent path as the data and so be constrained by the path width b.

Notice that there are no assumptions about the architecture of the parallel computer or the memory system. Thus, the memory might be local to the processor as in a multicomputer or it could be a global, shared memory as in a dancehall architecture.

Based on remarks in the Introduction stating that MIMD processing benefits from word parallel implementations, it might appear that bit serial is being discriminated against by the assumption that both addressing and data are constrained by b, but it is not. First, only the advantage of being "more direct" was claimed for word parallel and indeed the claim cannot be too strongly made since there are bit serial machines such as the CM2 [TMC 1987] permitting processors to perform independent addressing. Second, the results are little changed by ignoring the role of addressing, as explained later in the discussion of the D_{1m} case.

The data path width will in most cases be different from the word size. The *word size, w,* is the number of bits used to represent standard numerical quantities and addresses. Even though programs often process data that could be represented with fewer bits, general purpose computation does not usually exploit this fact. Similarly, machines often have less physical memory than their address space would permit, and even when they have that much, programs often process fewer data items than 2^w. Though these features would permit smaller addresses, the advantage is not usually exploited in general purpose computation and indeed "full addresses" are needed because of other encodings. Accordingly, it will be assumed that all data and address transmissions between processors and memory are for w size quantities. The specific benefits of operating on small data, which by the introductory remarks favors bit serial machines, will be considered later.

2.2 Computations

Informally, a computation is described by the reference behavior occurring on the data path between processors and memory. This activity may take the form of addresses presented to memory for fetching data or, if appropriate, fetching instructions, or it could be an address/value pair to be stored in memory. Thus, with each value there is an associated address, and so this will be the primitive unit of the model.

A *reference* is an ordered pair of nonnegative integers[1], $\langle a, v\rangle$, where a is called the *address* and v is called the *value*. A *thread* is an ordered pair, $\langle id, s_1 s_2 \cdots s_n\rangle$, where id is called the *identifier* and $s_1 s_2 \cdots s_n$ is a finite sequence of references called the *reference sequence*. (This usage borrows from analogous concepts in the Synchronous Taxonomy [Snyder 1988].) A thread is called a w-thread if for each reference $\langle a, v\rangle$ of the thread, $a < 2^w$ and $v < 2^w$, i.e. each address and value can be represented in a word.

A *computation task* is a set of threads with distinct identifiers; for convenience the identifiers of a computation task T will be taken to be the integers 1 to $|T|$. A computation is said to be a w-computation task if all of its threads are w-threads.

The computation task defines the set of processes (threads) to be performed by a parallel computer and each thread gives the sequence of memory operations, address/value pairs, that must be performed to implement the process. No operations are specified since they are largely irrelevant to the present analysis. Notice that the model is idealized in other ways. For example, the fact that the thread is specified by the sequence of the memory operations independent of the execution ignores such dynamic effects as a process busy-waiting on a semaphore. Such inaccuracies can be ignored provided they affect both wide and narrow machines analogously.

A w-thread, $\langle id_1, s_0 s_1 \cdots s_m\rangle$, is *simulated by* a u-thread, $\langle id_2, t_0 t_1 \cdots t_n\rangle$, where $du = w$ for some integer d, if, for each i, $s_i = \langle a, v\rangle$ implies $t_{di} = \langle a'_0, v'_0\rangle$, $t_{di+1} = \langle a'_1, v'_1\rangle, \ldots, t_{di+(d-1)} = \langle a'_{d-1}, v'_{d-1}\rangle$, and

$$a = a'_0 + a'_1 2^u + \cdots + a'_{d-1} 2^{(d-1)u}$$

$$v = v'_0 + v'_1 2^u + \cdots + v'_{d-1} 2^{(d-1)u}.$$

Clearly, the "simulated by" relation encapsulates the idea that w-sized quantities are encoded in d units of u-size. The definition performs the encoding by presenting the least significant unit first. This is clearly an inconsequential decision and the other encodings are equivalent. For convenience, the term *simulates* refers to the operands in the opposite order, i.e. a u-thread simulates a w-thread means the w-thread is simulated by the u-thread.

Proposition 1 *For any positive integers w, u and d, such that $w = ud$, every w-thread $\langle id_1, s_0 s_1 \cdots s_m\rangle$ is simulated by some u-thread $\langle id_2, t_0 t_1 \cdots t_n\rangle$. Moreover, $md = n$.*

[1]There is no loss of generality in ignoring various number representations.

Clearly, if u does not divide w, then w can be increased by an amount less than u to achieve divisibility and the proposition's truth is unaffected.

Let P denote the number of processors in a parallel computer. A *w-execution stream*, E, for a w-computation task T is a finite sequence of P-tuples, $p_1 p_2 \cdots p_n$, called *cycles*, where each cycle is of the form $\langle\langle id_1, ref_1\rangle, \ldots, \langle id_p, ref_p\rangle\rangle$, where id_i is a thread identifier from T and ref_i is a reference pair from thread id_i. It is further required that within each cycle the thread identifiers are distinct, i.e. $id_i \neq id_j$, that for each thread $\langle id_1, s_0 s_1 \cdots s_m\rangle$ each reference s_j appears once in an execution stream, i.e. there exists a k such that $\langle id_i, s_j\rangle$ is in p_k, and the references appear in order, i.e. if s_{j+1} is in p_l, $k < l$. The execution of a cycle is a step, so the time to execute an execution stream is n. The time of execution of the reference s_j is the time k of the cycle in which it appears. The concept to "simulate" extends to execution streams in the obvious way.

Notice that although the intuition of the model is that the first element of the cycle is the reference executed by the first processor, the second element is the reference executed by the second processor, etc., the positional order within the P-tuple is irrelevant; uniformly interchanging the i^{th} and j^{th} elements of a cycle anywhere in the execution stream will be regarded as an equivalent execution stream. Thus, without loss of generality, we will speak of *the* execution sequence and refer to some canonical representative. Even with this consolidation, there are many permissible execution streams for a given w-computation with more than P threads.

A thread $\langle id, s_0, s_1, \cdots, s_m\rangle$ in execution sequence E, in which $s_0 \in p_a$ and $s_m \in p_z$, is said to be *active* during cycle i provided, $a \leq i \leq z$. The times at which a thread begins, ends, or executes are a consequence of the algorithm's data dependencies, the compiler, the scheduler, and various dynamic features of the execution, which are beyond the scope of the present model.

The total number of active threads at time t of execution stream E is known as the *parallelism* Π_t. Clearly, this number varies, throughout the execution stream, as threads start and stop. However, it can always be determined in linear time in the length of the stream by counting the threads as they do so.

2.3 Latency

An execution stream has *latency* L if it satisfies the condition that for any two references from the same thread, the latter reference follows the former by at least L cycles. This modifies the sequentiality condition to read that given s_j and s_{j+1}, two adjacent references from thread id_i: with s_j in p_k, and s_{j+1} in p_l, then $k + L \leq l$.

It is implicit in this definition of latency that we are hiding latency by pipelining references. In fact, if we issue a large enough number of references from different threads, the latency will have no effect on the length of the execution stream.

On the other hand, to achieve a certain latency it may mean that it is not possible to schedule a full P-tuple for some cycle. If this is the case, it will be necessary to add null references to the cycle. A null reference is defined to be the pair $\langle 0, 0\rangle$ from a fictitious null thread numbered zero. The null thread is also exempt from the latency requirements, and may appear as often or as few times in the execution stream as is necessary.

If a particular cycle contains no null references, that cycle will be considered *saturated*.

By the above proposition, it is clear that any w execution stream can be simulated by some b-execution stream with time dilated by w/b. To account for differences in performance between machines, such as clock rate, the time dilation will be corrected by a constant C to be Cw/b, where $C = 1$ when the machines have equal memory reference rates.

3 Tradeoff Theorems

3.1 Saturation

The first theorem states that the point at which saturation occurs is at the same level of parallelism for any machine regardless of the number of processors and the data path width, so long as their product is held constant.

The crux of this argument is that the wider a processor is, the faster it executes each reference, but since the latency remains the same, the more references it must pipeline in order to hide that latency. On the other hand, while the narrower processor needs to pipeline fewer references, there are proportionately more processors, each of which needs its own queue of threads. These two requirements for parallelism balance each other, so the total parallelism required is the same; one processor of a given width requires twice as many threads to remain saturated as each of two processors of half that width.

The first part of the bound assumes that if there is sufficient parallelism, the machine is capable of scheduling any thread to any processor. This is essentially equivalent to stating that thread migrations can be performed at zero cost. The latter part follows directly from the delays required by the definition of latency.

Theorem 1 *A saturated execution stream can always be constructed if the level of parallelism is at least PbL/wC. Also, no saturated execution streams exist if the level of parallelism is less than PbL/wC*

Proof:
Part 1: Saturation possible if $\Pi_t \geq PbL/wC$. Consider the sequence

$$E = \left\{ \begin{array}{c} \langle\langle 1, s_{1,1}\rangle, \ldots, \langle P, s_{P,1}\rangle\rangle \\ \langle\langle 1, s_{1,2}\rangle, \ldots, \langle P, s_{P,2}\rangle\rangle \\ \vdots \\ \langle\langle 1, s_{1,Cw/b}\rangle, \ldots, \langle P, s_{P,Cw/b}\rangle\rangle \\ \langle\langle P+1, s_{P+1,1}\rangle, \ldots, \langle 2P, s_{2P,1}\rangle\rangle \\ \vdots \end{array} \right\}.$$

This sequence divides the active threads among the processors by associating thread *id* with the *id* mod P element of a cycle. Then a group of P w-references is simulated, for the corresponding Cw/b cycles of the b-stream. Now, each processor of the tuple will have at least bL/wC threads associated with it. Thus, when the last active thread is reached, at least the latency will have elapsed, since the time elapsed is $\Pi_t Cw/b > L$. At this point, it is now safe to resume execution with the next references of the first P threads without violation of the constraint on latency.

Part 2: No saturation if $\Pi_t < PbL/wC$. Consider a sequence of L successive saturated cycles in the execution stream. By the definition of L, none of the references in the elements of this sequence may be from the same thread. There are PL such references since there are P references for each of L cycles. Finally, each w-reference is simulated in Cw/b b-references. Thus, there must be at least PbL/wC threads active in this time. □

3.2 Simulation

The next theorem states that one execution stream can simulate another in real time, as long as they have the same product of data path width and number of processors. The reasoning behind this theorem is that by the preceding theorem, they saturate at the same level. The former stream can go no further than to the point of its saturation, and this is always within the limits of saturation of the latter stream.

Theorem 2 *Given a b_1-execution stream, E_1, for some w-computation task, there exists for each value of b_2, a b_2-execution stream E_2 that simulates E_1 such that if their values of L, C, and w are equal, and the products Pb are equal, then the length of E_2 is within the latency of the original length. That is,* $|E_2| \leq |E_1| + L$.

Proof:
We use induction on the time, taken in groups of L cycles, to construct an execution stream E_2 that always simulates at least as many w-references as E_1.

Basis: Initially, neither execution stream contains any references at time zero, so they have not simulated any w-references, either.

Induction Hypothesis: At a given time t, divisible by L, the two execution streams simulate an equal number of w-references.

We now prove that in the succeeding time interval of size L, they simulate an equal number of w-references.

Induction step: E_1 contains some number r of w-references in the cycles t to $t+L$. But, since these r references are all within time L of each other, they must all be from distinct threads. Thus, $\Pi_t \geq r$. On the other hand, there can be no more than PbL/wC references found in this interval, since that is the point of saturation, so $r \leq PbL/wC$. Thus $r \leq \min(PbL/wC, \Pi_t)$. But, by the construction found in the saturation theorem, we can find an execution stream E_2 which can execute r references in time L.

Thus, the latter stream can simulate all activities of the former performed in time L, in the same time L. In particular, the complete computation can be simulated in time $L\lceil t/L \rceil$, which is within L of the original time. □

A corollary to this result is that the optimal performance of both bit serial and word parallel machines is within L of each other, when simulating w-computation tasks, since each can simulate the optimal algorithm of the other.

Corollary 1 *Given two such machines M_1 and M_2, the optimal time on M_1, $T(E_1^o)$ is within L of the optimal time on M_2, $T(E_2^o)$. In other words, $|T(E_1^o) - T(E_2^o)| \leq L$.*

Proof: Let E_1^s be the execution stream produced by simulating E_2^o on M_1 using the theorem above. Similarly, let E_2^s be the simulation of E_1^o on M_2. Then $T(E_1^o) \leq T(E_1^s) \leq T(E_2^o) + L$ and $T(E_2^o) \leq T(E_2^s) \leq T(E_1^o) + L$ by the theorem above. This shows that $|T(E_1^o) - T(E_2^o)| \leq L$. □

4 Further Cases

4.1 SIMD Machines

In an SIMD machine, there is only one stream of instructions, but there are many data streams. The model does not distinguish in any way the purpose of a particular reference, merely its existence. Thus, the SIMD nature of a machine does not, *a priori*, change the results presented above.

The SIMD structure does, however, permit more variation in the generation of data addresses. In the one case of the D_{mm} structure, as is found on the CM-2 [TMC 1987] and Illiac IV [Bouknight *et al.* 1972], each processing element generates its own address, through a port of size b. In this case, the arguments of the two preceding theorems can be carried through entirely unchanged, save for noting that the reference streams are now *data* reference streams.

The alternative, the D_{1m} structure, only permits one global address, to which all processing elements must direct their accesses. In this case, the addresses are no longer part of the reference, eliminating the address portion of the half-cycle. On the other hand, the values transmitted continue to traverse the data path. The effects of this are twofold. Firstly, the word size w is bounded solely by the value, and not the address size. Since these quantities tend to be of similar magnitude, this effect, is, in general, unlikely to be significant. Secondly, one of the two half-cycles making up a cycle, the one associated with address transmission, is now unnecessary. While this factor of two is not large, it is a nontrivial time savings.

4.2 Small Latencies

So far, we have assumed the latency L to be a relatively long time, so that we use pipelining to hide that latency. If, on the other hand, the latency is less than the computation time, i.e. $L < Cw/b$, or $bL/wC < 1$, the computation time becomes the dominant term. Conversely, this is equivalent to slow instruction execution time, or a large value of C, since this affects the inequality similarly.

Under these conditions, we can return to our original definition of sequential execution, since successive w-references will always be simulated $Cw/b > L$ steps apart, or long enough that latency will always be preserved.

In this case, saturation occurs when the parallelism is at least as large as the number of processors, since there is no need to wait for latency. When saturated, each processor completes simulation of one w-reference in time Cw/b. Otherwise, in the same time, one w-reference is simulated for each of the active threads. We use the same construction as the original saturation theorem.

The four cases of saturation and latency are summarized as follows:

Theorem 3 *Given a computational task T, and an execution stream E, the average rate at which references complete, which is the ratio of the number of w-references in T to* $|E|$, *is no more than* $\min(Pb/Cw, \Pi_t/L, \Pi_t b/Cw)$.

Proof: Consider the four cases:

1. Latency bound, saturated. $L \geq Cw/b$ and $\Pi_t \geq PbL/wC$. In this case, the rate is the saturated rate of b/Cw per processor or Pb/Cw.

2. Compute bound, saturated. $L \leq Cw/b$ and $\Pi_t \geq P$. Here also, the saturation rate is Pb/Cw.

3. Latency bound, unsaturated. $L \geq Cw/b$ and $\Pi_t \leq PbL/wC$. In this case, we can make one reference per thread before waiting for the latency. Thus, the rate is Π_t/L.

4. Compute bound, unsaturated. $L \leq Cw/b$ and $\Pi_t \leq P$. In this case, we can only compute at full speed on each active thread, leaving the remaining processors idle. The rate is then $\Pi_t b/Cw$. □

4.3 Small word size

We now consider the case where $w < b$. In this case, some of the bits accessed in the data path will never contain any information relevant to the computation. One such case is the radix algorithm for computing the maximum, by examining each bit in order, from most significant to least significant. In this case, the data portion of each reference need only contain one bit.

Before we continue, it is necessary to point out that the word size w is the maximum of the address and data sizes. Thus, the word size will still be large if the addresses involved are large, even if the data values are only a single bit wide.

For instance, in the Connection Machine [TMC 1987], each of the processing elements operates on single-bit data, but the addresses are still of full size. In this mode, the processing elements may be spending as many as 32 cycles to generate the address for a single bit.

This is the analogue of the earlier wide-word D_{1m} case, where we are now reducing the number of value half-cycles, while leaving the address half-cycles unchanged. Once again, the maximum savings are limited to one-half.

We can, nonetheless, combine both of these, in the form of a bit-serial (or narrow-word) D_{1m} computational task.

If $w < b$, there is no need to simulate the wide access with a sequence of smaller accesses. Phrased another way, if $w < b$, any w-thread is, *a fortiori*, also a b-thread of the same length. In this case, the dilation of each reference attributable to the simulation is merely the constant C, which, in most cases, we take to be one.

Now, it is simple to determine whether the computation is latency or comput tion bound: the limiting factor is the larger of the two. In the compute bound case, saturation continues to occur at parallelism equal to the number of processors. When latency bound, each processor needs L/C references to remain saturated, for a total parallelism requirement of PL/C.

We can summarize these results for $w < b$ as showing that the performance rate is $\min(P/C, \Pi_t/L, \Pi_t/C)$. The combined result, for any combination of w, b, C, and L, is that the maximum performance is limited by $\min(Pb/Cw, \Pi_t/L, \Pi_t b/Cw, P/C, \Pi_t/C)$, which may also be written as $\min(\min(P, \Pi_t)\min(b/w, 1)/C, \Pi_t/L)$.

Theorem 4 *The performance of an execution stream is limited by the least of the following three factors:*

- *Simulation. No benefit arises from changing the data path size if it is already larger than the word size.*
- *Parallelism. No benefit arises from processors beyond the number of active threads in the computation.*

- *Latency. No benefit arises from any other change if all active threads are already executing once per latency time.*

The machine can execute at most $\min(\min(P, \Pi_t)\min(b/w, 1)/C, \Pi_t/L)$ *references per cycle.*

Proof: The theorem is a summary of the previous discussion. □

5 References

Bouknight *et al.* **1972** W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, A. H. Samek and D. L. Slotnick, "The Illiac IV System," *Proceedings of IEEE* 60(4): 369-388.

Batcher 1980 Kenneth E. Batcher, "Design of a Massively Parallel Processor," *Transactions on Computers*, C-29(9): 836-840, IEEE.

Danielsson 1983 P.E. Danielsson, "Algorithm-driven architecture for parallel image processing," in *Computer Architectures for Spatially Distributed Data*, Springer-Verlag, 1-17.

Hillis 1985 W. Daniel Hillis, *The Connection Machine*, MIT Press.

Hockney and Jesshope 1981 R.W. Hockney and C.R. Jesshope. *Parallel Computers*, Adam Hilger.

Kapauan *et al.* **1984** A. Kapauan, J.T. Field, D.B. Gannon, L. Snyder, "The Pringle Parallel Computer." *The 11th Annual International Symposium on Computer Architecture*, 12-21.

Motorola 1984 Motorola, *MC68020 Microprocessor Users' Manual*, Prentice-Hall.

Pfister *et al.* G. Pfister, W. Barntley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, S. Weiss. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture." *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE, pp.764-771.

Seitz 1985 C. Seitz. "The Cosmic Cube." *Communications of the ACM*, 28(1): 22-33.

Snyder 1988 L. Snyder, "A Taxonomy of Synchronous Parallel Machines." *Proceedings of the 1988 International Conference on Parallel Processing*, Penn State, pp.281-285.

TMC 1987 Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*. Technical report HA87-4.

Yang 1987 Chyan Yang, *An Investigation of Multigauge Architectures*, PhD dissertation. University of Washington TR 87-10-05.

Wirelisp: Graphical Programs for Digital System Design

Carl Ebeling and Zhanbing Wu

Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

The design of large, complex digital systems places an enormous burden on the systems architect. This is partially due to the crude tools still employed to specify, evaluate and implement large digital systems. In this paper we describe a language for describing systems that combines the immediacy and transparency of visual description with the expressiveness and generality of procedural description. These two forms of description are closely intertwined by embedding Wirelisp in Lisp and providing graphical constructs for most Wirelisp expressions. The power and flexibility of Wirelisp reduces the complexity of circuit specifications to a level consistent with the inherent complexity of the system being described.

Wirelisp is a language for representing the structure of a system as the hierarchical composition of elements, from primitive elements to complex modules. The concept underlying Wirelisp is that each device in the description is a procedure that understand how to construct instances of that device. Thus a specification may be implemented in a variety of ways depending on the application by simply changing how this construction is done. Arbitrary information can be included in the specification and used during this process.

Wirelisp is intended to be used to describe the structure of a system. However, some parts of a system are ill-structured and are better described behaviorally, that is, by describing the

relationship between input and output. Wirelisp allows such modules to be abstracted by means of their behavioral description, but the implementation of these modules must map the behavior into structure, a process left to synthesis tools external to Wirelisp. Wirelisp provides a framework within which behavioral information can be specified, evaluated in the context of the entire system and then implemented.

Wirelisp has been implemented using a generic drawing program along with an interpreter written in Lisp.

1 Introduction

Wirelisp is a language for describing the structure of digital systems as the hierarchical composition of elements. It is not the intent of Wirelisp to describe the *behavior* of a system, except indirectly as implied by the composition of primitive elements whose individual behavior is well-defined. In practice, systems are divided into modules that have obvious structure and modules that are better described by specifying their input-output behavior. Wirelisp allows such modules to reference a behavioral description, but does not support this type of description. At some point in the implementation, Wirelisp expects behavioral descriptions to be fully mapped to a structural description in Wirelisp. However, Wirelisp provides a hierarchical framework within which both the structure and behavior of a system can be described.

The benefits of hierarchy in the design of complex systems are well-known. It extends the designer's grasp by allowing him to abstract away the details at lower levels in the description. It reduces the interaction of system components by promoting modularity. And it reduces the size of a description by reducing the amount of repetitive and redundant information.

The structure of a system refers to the relationship between the elements of the system, both at the same level of hierarchy and between levels of hierarchy. That is, the structure specifies the connection between elements at the same level in the hierarchy, and the composition of elements from elements at the next lower level. These relationships are best represented graphically. It is much easier to draw the relationship between several items than describe it with words, and much easier to understand a relationship by seeing it, rather than reading a description of it. Thus circuits are typically described pictorally using schematics and there are today many schematic editors for describing circuits.

On the other hand, graphical descriptions lack flexibility and generality. Repetitive circuits are cumbersome to represent graphically, and similar but different circuits must be drawn separately, even if the differences between them are small. Graphical representations are also very "brittle". Simple changes in the system parameters can require very difficult and time-consuming modifications to the graphical specification. This problem is perhaps most evident in graphical layout systems. Graphical descriptions simply lack the expressive power to describe highly repetitive or parameterized circuits in a succinct way.

Procedural descriptions, by contrast, are much more powerful and flexible. Iterative constructs can be used to generate repetitive circuits. Conditionals can be used to modify the construction of a circuit based on parameters or iteration number. Even recursion is useful for describing certain types of circuits[5]. A procedural description, then, can represent an entire *family* of parameterized circuits, while a graphical description represents only a single instance.

Procedural descriptions, however, are difficult to write, and even harder to understand once written. Because of this they are typically used in a specify-evaluate-modify loop using some sort of visual feedback to inform the user of the structure implied by the specification. However, the power and generality of procedural descriptions often make up for their inpenetrability, and procedural descriptions have been found useful for many different tasks[4, 8].

Wirelisp incorporates the best of both approaches, allowing the user to interact graphically with the specification when describing relationships between elements while using procedural descriptions where flexibility and generality are required. Instead of merely allowing the user to "escape" to a programming language or providing a simple subset of procedural constructs, Wirelisp is embedded in Lisp, giving the designer the full power of a programming language. The designer is free to choose which type of description to use at each point in a design. Lisp expressions can be embedded in graphic descriptions and graphic descriptions embedded in Lisp expressions. This allows descriptions that are highly expressive, yet easily specified and understood.

Wirelisp programs are executed to produce a complete circuit description of some form. This might be a netlist suitable for board design, input to a simulator, or a description used to drive a layout generator. The actual form of the output representation is determined by the side effects generated by device procedures. For example, if each primitive device writes a description of itself to the output file along with the names of wires to which it connected, the result will be a flat netlist for the circuit. Libraries containing the procedures for the primitive devices are provided for common technologies such as CMOS. However, the user may write these procedures to effect whatever output is required.

2 Wirelisp Overview

In Wirelisp each device, or module, is a procedure. A device is instantiated by a call on this procedure. Because a device procedure may have parameters, it actually defines a family of devices, instances of which are produced by invoking the procedure with particular parameter assignments. One may view device procedures as entities that know how to construct instances of themselves.

Devices are either *composite* or *leaf* devices. Composite devices are comprised of other devices while leaf devices are primitive devices which have no components. Instances of composite devices are formed by invoking the device procedures of its components and connecting the resulting instances together. Devices are connected together via signal wires attached to connection points on the devices. Connection points, which correspond to device input and output ports, are represented by signal parameters. When a device is instantiated, the names of the wires to which it is connected are passed as signal parameters to the device procedure.

Signals correspond to wires and are objects in their own right. Signals can be defined to have structure, that is, composed hierarchically of other signals. Both homogenous structures (busses) and heterogenous structures (records) can be specified. The use of structured signals increases the level of abstraction used in a description and decreases the size of the graphical description by bundling several related signals into one signal.

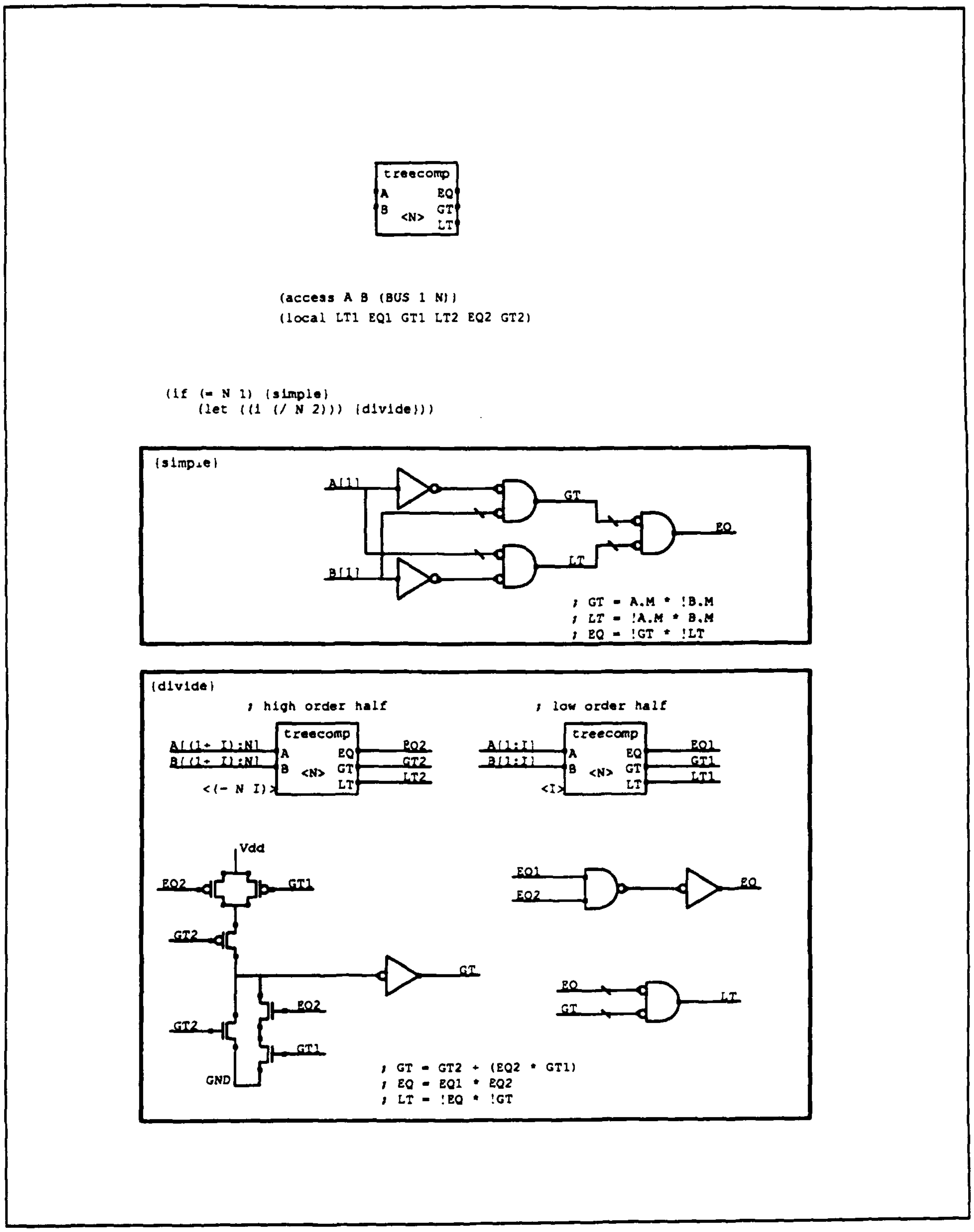
Wirelisp device procedures are drawn rather than written. A composite device is defined by giving the drawing symbol for the device, which contains both signal and general parameters, and all the component devices and their interconnection. Behind this graphical representation lies a Lisp procedure. Lisp expressions used in the drawing appear in this procedure verbatim and are evaluated when the device procedure is invoked. An example of a device definition is given for a device called treecomp in Figure 1. This device compares two N-bit numbers via a recursive divide and conquer approach.

Leaf devices correspond to the primitive devices in whatever technology is being used. When a hierarchical circuit description is fully elaborated, all that remains is the entire set of interconnected leaf devices. This flat circuit description is the most common result of executing a Wirelisp program, and is produced by programming each primitive device procedure to write a description of an instance of itself to the output file. For common technologies and output file formats, these primitive device procedures would be in a library. Although this is the most common result of executing Wirelisp programs, the user is free to redefine the actions taken by both primitive and composite devices to produce other types of output.

3 Related Work

The idea of using a programming language to represent circuits has been used previously, notably the use of Lisp in **netlist** [11] and DPL[4]. Wirelisp is similar to **netlist** in its use of procedure hierarchy and its definition of device connections via parameters. However, **netlist** lacks any graphical representation, has only a very primitive notion of signal structure that prevents information hiding, and is very difficult to target to different applications.

The use of hierarchy in the description of circuits has been used for a long time. The Stanford



```
(access A B (BUS 1 N))
(local LT1 EQ1 GT1 LT2 EQ2 GT2)

(if (= N 1) (simple)
    (let ((I (/ N 2))) (divide)))

; GT = A.M * !B.M
; LT = !A.M * B.M
; EQ = !GT * !LT

; GT = GT2 + (EQ2 * GT1)
; EQ = EQ1 * EQ2
; LT = !EQ * !GT
```

Figure 1: *A Wirelisp Drawing*

University Drawing System (SUDS) was the first widely used drawing system that incorporated the idea of macros to represent entire subcircuits. Most drawing systems today allow hierarchical specification; however, they all share the relative inflexibility of graphical descriptions.

Both graphically and procedurally based VLSI layout systems have been used, but little has been done to incorporate both types of description in the same specification. Some systems such as Mocha Chip[9] and WIN[1] use both types of description in the specification of chip layout, but they cannot be intermixed at a fine-grained level.

Xerox PARC has recently produced a circuit specification system that incorporates both graphics and programs[2, 3] and is similar in philosophy to Wirelisp. However, Wirelisp intertwines the graphical and procedural constructs more closely which yields specifications that are more descriptive and self-apparent.

4 Device Definition and Instantiation

Devices are defined by Wirelisp procedures. Composite devices describe their component devices and their interconnection by invoking the procedures corresponding to those devices with the appropriate parameters. Primitive devices do not invoke any devices but perform actions such as writing the appropriate description of that device to an output file.

A device procedure is typically defined by a single drawing as in the example of Figure 1. The device symbol being defined is placed at the top of the page and corresponds to the procedure head. The interconnected component devices are placed below it and form the body of the procedure. The device symbol contains named connection points for the input/output signals of the device. These correspond to the signal parameters of the device. Pins are used to mark the connection points in the device symbol, and the device definition provides the parameter name for each pin. General parameters, both mandatory and optional, are simply listed along with the device symbol enclosed by the special chararacters < and >.

The device symbols appearing in the procedure body each corresponds to a procedure call. The names of the wires connecting these symbols via connection points are passed as the corresponding actual parameters to these procedure calls. The actual general parameters for device instances are listed next to the device symbol. Optional parameters are named; that is, they are not specified by position as are mandatory parameters, but as a (name value) pair. Device instances may specify optional parameters that do not appear in the device definition, in which case they are ignored. While this may not appear to make sense, it allows the designer to attach arbitrary information to a device instance. Only if this information is understood by the device definition is it used. For example, the designer can include layout information which is ignored by a definition that generates netlist.

The signal wires appearing in the procedure body must either be a formal parameter, i.e. an input or output signal, or be declared as a local signal. If a wire does not have a name, then it is automatically declared as a local signal and given an arbitrarily derived name. Defining signals locally allows irrelevant information to be hidden from the surrounding context. Signal wires are represented as simple lines in the drawing and are named by closely placed text. Different wires with the same name are considered to be the same wire, and naming a wire more than once creates alias names for that wire.

Lisp expressions may be used anywhere in a drawing and are included in the device procedure as they appear in the drawing. The most commonly used Lisp expressions are conditionals, loops and arithmetic operations; however, any Lisp function can be used, including those defined by the user. Conversely, circuit drawings may be used within Lisp expressions. This is done by enclosing the circuit drawing with a named rectangle and referencing this name in the Lisp expression. Figure 1 contains an example of this. The if conditional is used to generate one of two circuits depending on the size of the input values.

Lisp expressions are evaluated in the order in which they appear on the page, although typically their order is irrelevant since circuit drawings are declarative in nature. The closely entwined relationship between graphics and Lisp expressions means that there are no *ad hoc* features needed to fill gaps in the expressiveness of the language. The entire functionality of Lisp is available just beneath the surface whenever it is needed.

Branch points can occur in the hierarchy if alternative implementations exist for a device. Design alternatives are easily implemented in Wirelisp by using the analog of a case statement in the device definition that chooses the implementation based on a global variable. For example, one might have a `style` variable that is assigned the value `simulation` or `netlist` depending on whether the output is to be used by a simulator or is simply a flat wirelist. In the case of simulation, a composite device may be implemented as a functional model while in a netlist, its details would be fleshed out.

The result of executing a Wirelisp program can, in principle, be defined by the user. However, the user will typically use library procedures for a particular technology and a standard output representation of the circuit. The most common use of Wirelisp produces a flat representation of the circuit containing only primitive devices connected as specified in the Wirelisp program. For example, at the University of Washington we use Wirelisp to design CMOS circuits and produce output in standard SIM format. However, the Wirelisp program might just as well build an in-core representation of the circuit or make entries directly into a design database like Berkeley OCT[7]. The flexibility of a procedurally based description allows the general Wirelisp framework to be used for a wide variety of applications with only a small amount of additional work.

5 Structured Signals

At higher levels of abstraction in a hierarchical description, modules typically operate on values that are more complex than simple signals. Examples include busses, numbers, and control words. We refer to a collection of related signals generically as a *bundle*. A bundle comprises an ordered set of signals which can be referenced collectively by the name of the bundle, or individually via an *access mechanism.* Bundles are created via a local declaration which declares the bundle name, the component names, and creates the *physical structure* of the signal. Alternatively, bundles can be created dynamically by binding a list of already declared signals using the bundle operator. This only creates a physical structure without declaring any new signal names.

Bundles are allowed to have hierarchical structure. That is, a bundle can be defined recursively:

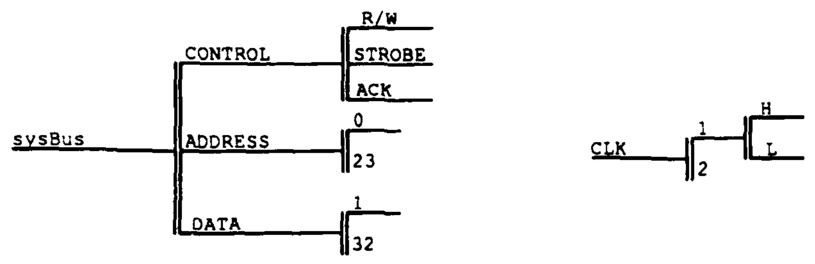
1. A simple signal is a bundle.

2. An ordered set of bundles is a bundle.

A hierarchically structured bundle is then an ordered tree in which each leaf node is a simple signal and each interior node is a structured bundle.

Wirelisp provides two different ways to access structured signals: *records* and *busses*, corresponding to records and arrays in conventional languages. Records use named selectors, which are compile-time constants, to access elements in a bundle while busses use indices, which may be variable. Wirelisp separates the physical structure of a signal from the access mechanism used to access its components. Thus a structured signal may be referenced as a bus in one device and a record in another. Typically, however, collections of homogenous signals are defined as busses and collections of heterogenous signals as records.

Access structures may be defined either textually or graphically as shown in Figure 2 according to the designer's taste. Access structures are used in a straightforward way to access any node



```
(sysBus (SN (CONTROL (SN R/W STROBE ACK))          (CLK (BUS 1 2 (SN H L)))
            (ADDRESS (BUS 0 23))
            (DATA (BUS 1 32))))
```

Figure 2: *Graphical and textual declarations of structured signals.*

in the structure tree. For example, **sysBus.control** refers to the three signals **R/W**, **STROBE**, and **ACK**, **sysBus.ADDRESS[23]** refers to the high-order address bit, and **CLK[1]** refers to both assert high and low signals of the first phase of a clock.

A design typically uses only a few different access structures, one for each type of bundle used in the design. If every device were required to define the access structure, it would be difficult to keep structure declarations consistent. It would also require substantial effort to change the declarations when bundle structures are changed or refined. Wirelisp implements global *structure types* which allow common structure definitions to be defined once and then used throughout the design. By using these global definitions, every device is assured of having a consistent view of the world.

When a bundle is passed as a parameter to a device, an access structure must defined in order to access its components. This access structure tree must match the physical structure tree in the following recursive sense, starting at the root:

1. Two interior nodes match if they have the same number of children.

2. An access leaf node may match an physical interior node.

3. An access leaf node matches a physical leaf node.

This extends the abstraction inherent in structured signals by allowing devices to access only that part of the signal they use, leaving the rest hidden. Wirelisp dynamically checks to make sure that the structure used to access a bundle matches the actual physical structure of that bundle.

Bundle declarations and expressions often contain long lists of similar names. Wirelisp provides a simple mechanism, called the *iterator*, for describing such lists of names succinctly. Iterators are merely syntactic shorthand used to reduce the amount of repetitive information in a drawing. An iterator occuring in a name causes that name to be expanded into a list of names according to the iterator. An iterator has the form `n1:n2`, where `n1` and `n2` are maximal substrings of digits appearing in the name string. For example, `data16:31H` contains the iterator `16:31`. The list of names generated consists of one name for each value in the range `[n1...n2]` where the iterator is replaced by its value. For example, `data16:31H` expands to the list `data16H data17H data18H ... data31H`.

An arbitrary number of iterators may appear in a name string, as in `bit0:7-0:3`. This forms a nested loop, with the outermost loop to the left. This example would expand to the list `bit0-0 bit0-1 ... bit7-3`. If necessary, an iteration increment can be specified by an iterator in the form `n1:n2:n3` where n3 is the increment. Also, `n1` may be greater than `n2`, in which case the increment is assumed to be negative.

6 Naming Issues

All signals and device instances in a circuit are distinguished by assigning them unique names. These names then appear in the output when signals or device instances are referenced. The user has two choices for the form of these names: hierarchical or abbreviated. Hierarchical names indicate precisely where in the description hierarchy a signal was declared or a device instance created.

Device instance names are optional parameters which default automatically to the device name plus a small integer that distinguishes devices of the same type within one instance. This instance name is unique only in the context of the parent instance. A unique, hierarchical instance name is created by taking the pathname comprising all instance names from the root to the instance. That is,

1. The pathname for the root device is '/'.

2. The pathname for all other devices is 'parent-pathname/instance-name'.

Hierarchical signal names are formed by concatenating the pathname of the instance in which the signal was declared with the signal name.

For large circuit these hierarchical names can become very long, making the output files unreasonably long. In this case the designer may choose to use abbreviated names for signals and instances which do not need to be observed in the output. Abbreviated names for instances and signals are formed by concatenating the simple signal or instance name with a globally unique integer. For example, the `slocal` declaration is used to declare signal names that should be abbreviated.

7 Including Behavioral Information in Wirelisp

While much of the description of a digital system is structural, there are parts of the system which have no obvious structure, but whose input-output behavior is well-understood. While such parts must eventually be mapped into an implementation whose structure can be described using Wirelisp, it is not the intention of Wirelisp to perform behavioral synthesis tasks except by invoking other tools. But by having the behavioral part of the system integrated with the structural description, the system can be designed, evaluated and implemented as a whole. For example, as an architectural description is being evaluated using simulation, the description of the behavioral modules can remain at the abstract behavioral level and simulated functionally.

This inclusion of behavioral information in a Wirelisp description is relatively straightforward. A module is created with the appropriate input-output signals and with an implementation that refers to the behavioral description, which may be in whatever form the user desires. For example, this behavior may be given as a set of logic equations, a finite state machine description in a high-level language, or a state diagram. The only requirement of Wirelisp is that this behavioral description be transformed by some agent into a form compatible with the output description of the rest of the circuit. For example, a PLA description might be translated into a model for simulation or into a Wirelisp description of a PLA circuit implementation depending on a global style variable.

Following the view of devices as entities that know how to construct instances of themselves, behavioral modules contain both a behavioral description and a method for constructing the appropriate output description from that behavior. This may be as simple as compiling a FSM and generating the PLA output or so complicated that the designer must intervene to generate the corresponding structure description.

An example of a very simple version of behavioral description is the ability to use logic equations in a drawing. We have extended Wirelisp to allow the inclusion of simple logic equations as replacements for small collections of logic gates. For example, in Figure 1 the logic gates computing `EQ`, `GT`, and `LT` could be replaced by logic equations describing the function they compute. This has been implemented by a user Lisp function that converts a logic equation into a device that implements that equation. While in principle logic or behavioral synthesis could be made part of the Wirelisp environment, in practice it makes more sense to allow Wirelisp descriptions to take advantage of the full range of tools available for this problem.

8 Physical Descriptions

Wirelisp describes only the topological structure of a system which, for some technologies, is insufficient for implementation. *Physical* information is required in addition to the topological information. The best example is custom VLSI chip layout where there are many ways to physically implement the same logical circuit, but determining this information from the topological structure is extremely difficult. Wirelisp is not intended as a layout description language; however, it allows sufficient physical information to be added to a structural description to allow a backend layout system to generate the chip layout. The amount of additional information required depends on the sophistication of the backend system and the quality of the layout desired.

Physical design can be viewed hierarchically as well. Layout leaf cells are composed to build larger modules which are themselves composed into still larger modules. Wirelisp allows the designer to include the information to perform this type of physical design. Devices procedures corresponding to the layout leaf cells refer to the layout, presumably generated using a layout editor, but possibly generated automatically, for example as standard cells. Examples of layout leaf cells include register cells, ALU bit-slice, shifters, memories and standard cell implementation of random logic. These occur higher in the hierarchy than the primitive logical devices and thus a design will be instantiated in different ways depending on the intended output.

The composition of layout modules is specified by describing the relative position of these modules. This information can be given either graphically or textually depending on what is appropriate. For example, devices can be placed in the drawing corresponding to their relationship in the layout, or, if this makes the drawing awkward, their physical relationship can be described textually.

Given this information in the Wirelisp description, the physical design system acts as a chip assembler, composing the layout cells hierarchically according to the physical information in the Wirelisp description. While this provides a front-end to the chip assembly task, it does not address the difficult problem of designing the cells so that they fit together properly at all levels of the hierarchy. In practice, cells within regular modules such as register files, memories and data paths will be designed to fit by abutment. Modules are then wired automatically after being placed according to the Wirelisp description.

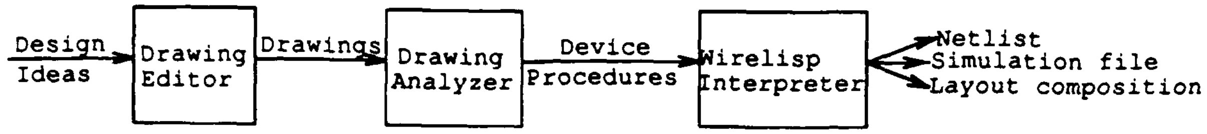


Figure 3: *Wirelisp Impementation*

The hierarchy of the logical and physical description of a circuit are often different. For example, a datapath might logically be described as the interconnection of an ALU, shifter, and latches, while physically it would be viewed as the composition of several bit-slices. Wirelisp allows divergent and reconvergent hierarchies through the use of case expressions based on style that direct the implementation along differing paths.

We see Wirelisp as a way to implement module generators for a wide variety of structured modules. Of course one wants more intelligent tools which can deduce the physical structure from the topological structure, relying on the architect to have specified a design amenable to layout in the plane. We believe that Wirelisp would be an appropriate language for describing the input to such a silicon compilation system.

9 Wirelisp Implementation

There are a range of possible implementations of Wirelisp depending on how closely the drawings and their interpretations are bound. We describe here the current implementation of Wirelisp which uses a generic drawing program and an interpreter written in T Lisp[10]. Figure 3 shows the flow of information from concept, to specification, to implementation.

Wirelisp drawings are made using DP[6], a generic drawing editor from Carnegie -Mellon University that runs on Sun and Microvax workstations. DP supports a variety of graphic objects including lines, text, circles, arcs and spline curves. DP provides some help with circuit drawing with respect to connecting lines used as wires, but otherwise does not understand the semantics of circuit drawings. A backend program which does understand these semantics converts Wirelisp drawings into Wirelisp procedures. The designer, however, only interacts with the circuit specification via the drawings.

DP has a macro concept whereby a group of related objects can be bound into a single named object called a symbol. Symbols in Wirelisp drawings correspond to devices, and connection points within devices are represented by DP pins around the periphery of the device. Making a device definition involves defining a symbol for the device, copying in the symbols of the component devices from the drawings in which they are defined, and then connecting them using lines attached to their pins.

DP allows the designer to move about in the hierarchy of a design by "subediting" devices. This causes the current drawing to be suspended and replaced by the drawing for the device being subedited. Exiting then causes the editing of the suspended drawing to be resumed. Subediting is allowed to any depth.

The execution of Wirelisp procedures resulting from the analysis of the drawings is performed by an interpreter written in T Lisp. This is done by entering the interpreter and giving it the name of the root device. The interpreter simply executes the procedure for the root device. As subsequent device procedures are called, drawing programs are automatically reanalyzed if they have changed and device procedures are autoloaded. The current implementation executes Wirelisp programs at the rate of about 200 devices/second for moderately complex CMOS circuits on a Sun 3. This means that even large circuits with 100,000 transistors take only about 10 minutes to process.

10 Conclusion

Wirelisp is a highly expressive language for describing complex digital systems. This expressiveness derives from the close relationship between graphical and procedural descriptions. Wirelisp provides a framework within which a variety of different implementation information can be specified. This allows multiple implementations of a specification, ranging from a netlist, input to a functional simulator and layout.

Future plans for Wirelisp include writing functions that make it easier to incorporate behavioral descriptions and investigating ways to utilize partial information in the structural description to generate layout. This will produce a very flexible and easy way to perform module generation and might be extended to the composition of layout modules.

References

[1] J-L. Baer, M. Liem, L. McMurchie, R. Nottrott, L. Snyder, and W. Winder. A notation for describing multiple views of vlsi circuits. In *Proceedings of the 25th Design Automation Conference*. IEEE, June 1988.

[2] R. Barth and B. Serlet. A structural representation for vlsi design. In *Proceedings of the 25th Design Automation Conference*. IEEE, June 1988.

[3] R. Barth, B. Serlet, and P. Sindhu. Parameterized schematics. In *Proceedings of the 25th Design Automation Conference*. IEEE, June 1988.

[4] J. Batali, N. Mayle, H. Shrobe, G. Sussman, and D. Weise. The dpl/daedalus design environment. In *VLSI '81*, pages 183–192, 1981.

[5] Edmund M. Clarke and Yulin Feng. Escher-a geometrical layout system for recursively defined circuits. *IEEE Transactions on Computer Aided Design*, 7(8):908–918, August 1988.

[6] D. Giuse. Dp: A drawing program. Technical report, Computer Science Dept., Carnegie -Mellon University, 1982.

[7] P. S. Harrison, P. Moore, R. L. Spickelmier, and A. R. Newton. Data management and graphics editing in the berkeley design environment. In *Proceedings of ICCAD*, pages 24–27. IEEE, November 1986.

[8] C. Kingsley. *Earl: a language for integrated circut design.* PhD thesis, California Institute of Technology, 1981.

[9] Robert Mayo. Mocha chip: A system for the graphical design of vlsi module generators. In *Proceedings of the ICCAD*, pages 74–77. IEEE, November 1986.

[10] Stephen Slade. *The T Programming Language: A Dialect of Lisp.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[11] Christopher J. Terman. *Simulation Tools for Digital LSI Design.* PhD thesis, Massachusetts Institute of Technology, October 1983.

Ohmics: A Program to Check Ohmic Contacts

Wayne E. Winder

29 March 1989

1 Introduction

In CMOS circuits, ohmic contacts must be near transistors to correctly bias base voltages and avoid the latch-up phenomenon. Designers using *magic* [1] to design circuit layouts must place these contacts manually.

Since ohmic contact placement is done manually, four types of errors can be made. First, an ohmic contact can be connected to the wrong supply rail. Second, the contact material can be of the wrong type. Third, since *magic* creates well regions automatically, it is possible for the designer to unwittingly isolate transistors in their own wells, with no ohmic contact present. Finally, it is possible to place the closest ohmic contact too far from the device (this can also be viewed as a failure to place a contact).

Ohmics has been developed to detect these errors.

2 Design Environment

N-type (p-type) transistors must be near p-ohmic (n-ohmic) contacts connecting the p-substrate (n-substrate) to ground (Vdd). A check for this condition must be done on extracted layout to determine the voltages of the ohmic contacts. In order to detect intervening material of the opposite type substrate, this check must include tracing the connectivity through the substrate. *Mextra* [2] has been modified to trace this connectivity, to stamp all transistors with the node name of the containing substrate, and to output the location and node name of all ohmic material.

Ohmics processes one neighborhood (called a frame) of the chip at a time. All the data for the frame must be in memory, so the frame size is limited. While very large (nominally 1800 lambda), one frame may be smaller than the design. To avoid running out of memory and to increase efficiency, a new filter, *simsort* has been added to sort the extracted data by the y-coordinate of the transistor location.

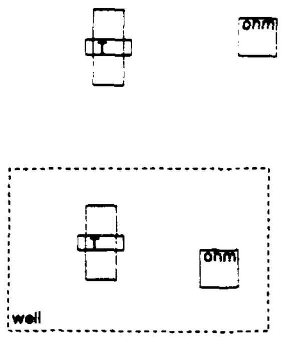


Figure 1: Input Data

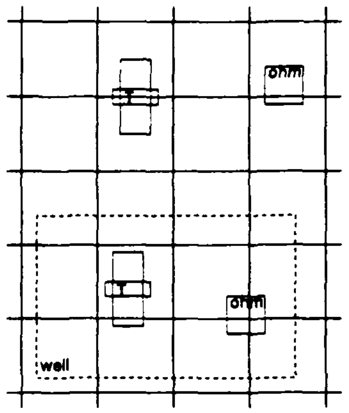


Figure 2: Instantiate Data

Ohmics is run on the sorted data to detect the errors. For each transistor, *ohmics* searches for the closest ohmic contact of the correct type that is connected through the appropriate substrate. *Ohmics* outputs text and graphic data locating problem areas. The graphic data is converted to *magic* form by the new filter *showohms*. In this form, the designer can locate the errors visually on the *magic* display.

3 Algorithm

The input to *ohmics* is substrate geometry from the *cif* file, the locations and node names of p- and n-ohmic contacts and the locations and substrate nodes of p- and n-type transistors (see Figure 1).

One frame is processed at a time with the algorithm repeated on each frame. The data is instantiated into the frame which is cut into a grid. Contacts and transistors are assigned to the grid squares in which they lie and for the remainder of processing are considered to be in the middle of their squares. Substrate boundaries are found by intersecting grid corners with substrate geometry. (See Figure 2) All subsequent processing is done on grid squares rather than the exact location of items in the grid.

Next, the substrate corners are marked. (This also marks the well corners, since they are the same.) This must be done after instantiation since the cif geometry is in rectangles and therefore corners cannot be known until all the geometry has been read. (See Figure 3)

For each transistor, a spiral search outward is performed until the closest directly connected (DC: no substrate boundary along the line connecting the two) ohmic contact is found up to the maximum search distance. All non-DC ohmic contacts and all substrate corners found are saved for possible use later.

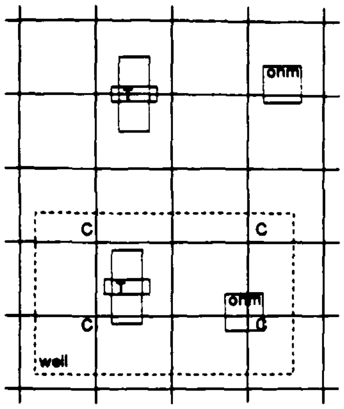


Figure 3: Mark Corners

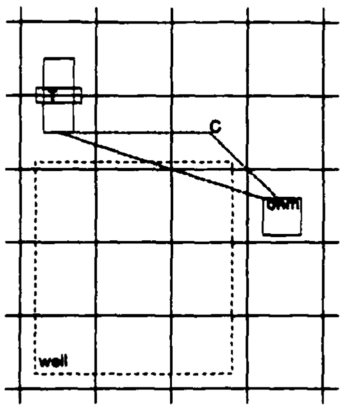


Figure 4: Corner Search

If there is no DC ohmic contact or a non-DC ohmic contact is closer than the DC ohmic contact, a corner search is performed using the data previously saved. (See Figure 4) This search requires an iteractive sorting of the saved corner data by distance from the device, adding the closest corner to the active list on each iteration. This is a slow approach, but experiment shows that corner search is required in less than 5% of the total number of searches.

If no ohmic contact was found or an ohmic contact was found that was beyond the distance threshhold, *ohmics* outputs error data. This data gives the location of the transistor and the closest contact, if it exists, which is beyond the threshhold. Also output is graphic data which can be filtered into a new magic file (with *showohms*) to visually display errors.

4 Performance

Ohmics is based on *mextra* code and performs a subset of its extraction. It therefore has execution times of the same order as extraction of layout. While these times are large for entire chips (such as APEXII), it is expected that designers will check individual modules prior to chip assembly. Ohmic contact problems can only be introduced at higher levels of the hierarchy by miswiring (ground and Vdd) or by excessive overlap of well regions. In the majority of cases, few errors will be introduced during chip assembly.

In the following table, *pe* is an 8k transistor processing element in APEXII, a 50k transistor curve generation chip. This approximates the largest module that can be expected in most cases. *Pe2* is *pe* rotated 90 degrees.

design	tran- sistors	user (sec.)	sys (sec.)	clock (min.)
pe	8k	142.4	5.4	3
pe simsort		63.0	4.0	1
pe2	8k	138.0	10.5	3
pe2 simsort		59.0	5.3	1
APEXII	50k	826.2	123.8	29
APEXII simsort		310.8	23.4	7

5 Conclusion

We have created a tool, *ohmics*, which will tell the designer if ohmic contacts are connected to the wrong supply rail, if the wrong type of ohmic material was used in a well (substrate), if no ohmic contact was placed within some distance of an ohmic contact and if *magic* has created a well region which is isolated from all ohmic contacts.

References

[1] G. Hamachi, R. Mayo, J. K. Ousterhout W. Scott, G. Taylor. *Magic - VLSI layout editor* from 1986 VLSI Tools, Computer Science Division, U. C. Berkeley, December 1985

[2] Dan Fitzpatrick *Mextra - Manhattan circuit extractor* from 1983 VLSI Tools, Computer Science Division, U. C. Berkeley, March 1983.